UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
10/630,913	07/31/2003	Venugopal K. Srinivasamurthy	200313704-1	7570

22879 7590 05/12/2010

HEWLETT-PACKARD COMPANY
Intellectual Property Administration
3404 E. Harmony Road
Mail Stop 35
FORT COLLINS, CO 80528

EXAMINER	
DAO, THUY CHAN	

ART UNIT	PAPER NUMBER
2192	

NOTIFICATION DATE	DELIVERY MODE
05/12/2010	ELECTRONIC

Please find below and/or attached an Office communication concerning this application or proceeding.

The time period for reply, if any, is set in the attached communication.

Notice of the Office communication was sent electronically on above-indicated "Notification Date" to the following e-mail address(es):

JERRY.SHORMA@HP.COM
ipa.mail@hp.com
laura.m.clark@hp.com

UNITED STATES PATENT AND TRADEMARK OFFICE

Commissioner for Patents
United States Patent and Trademark Office
P.O. Box 1450
Alexandria, VA 22313-1450
www.uspto.gov

BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES

Application Number: 10/630,913
Filing Date: July 31, 2003
Appellant(s): SRINIVASAMURTHY ET AL.

John P. Wagner
For Appellant

EXAMINER'S ANSWER

This is in response to the appeal brief filed February 18, 2010 appealing from the Office action mailed October 22, 2009.

(1) Real Party in Interest

A statement identifying by name the real party in interest is contained in the brief.

(2) Related Appeals and Interferences

The examiner is not aware of any related appeals, interferences, or judicial proceedings which will directly affect or be directly affected by or have a bearing on the Board's decision in the pending appeal.

(3) Status of Claims

The statement of the status of claims contained in the brief is correct.

(4) Status of Amendments After Final

The appellant's statement of the status of amendments after final rejection contained in the brief is correct.

(5) Summary of Claimed Subject Matter

The summary of claimed subject matter contained in the brief is correct.

(6) Grounds of Rejection to be Reviewed on Appeal

The appellant's statement of the grounds of rejection to be reviewed on appeal is correct.

(7) Claims Appendix

The copy of the appealed claims contained in the Appendix to the brief is correct.

(8) Evidence Relied Upon

6,988,261	Sokolov et al.	01-2006
6,014,519	Egashira	01-2000

(9) Grounds of Rejection

The following grounds of rejection are applicable to the appealed claims:

❑ Claims 1 and 21-39 are rejected under 35 U.S.C. 103(a) as being unpatentable over Sokolov (US Patent No. 6,988,261) in view of Egashira (US Patent No. 6,014,519).

Claim 1:

Sokolov discloses *a method of optimizing the performance of an interpreter based runtime system, the runtime system including a virtual machine, the virtual machine adapted to run an application in the context of the runtime environment, the method comprising:*

augmenting a bytecode set of the virtual machine with semantically enriched opcodes (e.g., FIG. 12A, conventional Java bytecode "Getfield" and "Astorex" are augmented/replaced by inventive opcode "Get_Store" (a semantically enriched opcode); FIG. 13C, conventional Java bytecode "lastore" and "dastore" are augmented/replaced by another inventive opcode "ASTOREL" (another semantically enriched opcode"; FIG. 2B, col.6: 55 – col.6: 26).

thereby constituting an application domain-specific virtual machine (e.g., col.4: 3-15, col.6: 23-26, an augmented/modified Java virtual machine specific to the inventive opcodes, col.7: 51-59; FIG. 2A-B, col.8: 6 - col.9: 17, an augmented/modified Java virtual machine specific to the inventive Java macros);

optimizing the virtual machine based on semantics of the application to be run on the virtual machine (e.g., FIG. 5, col.7: 37-67; FIG. 2B-C, col.8: 62 – col.9: 17),

with at least a portion of the semantically enriched opcodes being specific to the application (e.g., FIG. 4, col.7: 6-19, block 404-406, each application has different frequently repeated bytecode → block 408, specific macro instructions are generated, i.e., semantically enriched opcodes specific to each application);

performing a quantitative comparison between execution time and memory space (e.g., col.2: 64 – col.3: 9 and col.5: 66 – col.6: 9, the inventive opcodes

executes faster (fewer instructions) and occupy less memory space (fewer instructions); col.8: 1-11)

to determine effective semantically enriched opcodes and encoding the semantically enriched opcodes into interpreter action codes based upon the comparison (e.g., col.3: 14-col.4: 15 and FIG. 4, after reaching a predefined threshold, generating Java macro instructions (inventive opcodes/semantically enriched opcodes) based on the comparison (executing faster and less memory space); col.5: 28 – col.6: 38; col.11: 9-32)*;*

analyzing frequently executed bytecodes (e.g., FIG. 4, col.6: 66 – col.7: 19) *and*

encoding the semantically enriched opcodes into interpreter action codes of the instruction set of the virtual machine to efficiently decode the frequently executed bytecodes (e.g., FIG. 9B, col.9: 50 – col.10: 11)*;*

optimizing the encoding the semantically enriched opcodes into the interpreter action codes according to a system state, said system state being represented by at least one symbolic variable (e.g., FIG. 4, block 406 "Sequence has been counted for at least a predetermined number of times? YES → block 408 "Generate a Java macro instruction that represents the sequence of Java Bytecode instructions", i.e., only generates macro instruction and passes it to Java interpreter based on whether a counter reaches/exceeds a predefined threshold (a predefined system state), wherein said predefined threshold (said predefined system state) is a variable and compared with a counter – see block 404 "Count the number of times …", emphasis added)*; and*

statically embedding the semantically enriched opcode to optimize execution of the interpreter-based runtime system (e.g., FIG. 4, block 408, col.6: 66 – col.7: 19; FIG. 5, col.7: 37-67)*.*

Sokolov does not explicitly disclose *performing a quantitative trade-off between execution time and memory space to determine effective semantically enriched*

opcodes and encoding the semantically enriched opcodes into interpreter action codes based upon the trade-off.

However, in an analogous art, Egashira further discloses *performing a quantitative trade-off between execution time and memory space to determine effective semantically enriched opcodes and encoding the semantically enriched opcodes into interpreter action codes based upon the trade-off* (e.g., FIG. 3, col.7:60 – col.8: 53; FIG. 7, col.12: 57 – col.13: 18; col.6: 60 – col.7: 33).

It would have been obvious to a person having ordinary skill in the art at the time the invention was made to combine Egashira's teaching into Sokolov' teaching. One would have been motivated to do so to generate an object module file with a shorter execution time in a range of code size as suggested by Egashira (e.g., col.2: 45-54; col.3: 22-29; col.3: 62 – col.4: 39).

Claim 21:

Sokolov discloses *the method of claim 1, further comprising analyzing an application code using static optimization, the static optimization comprising parsing the application code to identify at least one repeated sequence of bytecodes* (e.g., FIG. 4, block 402 "Read a stream of Java Bytecode instructions ..." → block 404 "Count the number of times ...", col.2: 19 – col.3: 47, i.e., parsing and identifying at least one repeated sequence) *and*

replace the at least one repeated sequence of bytecodes with a semantically enriched opcode (e.g., col.5: 28 – col.6: 38).

Claim 22:

Sokolov discloses *the method of claim 1, further comprising analyzing an application code using dynamic optimization, the dynamic optimization comprising: analyzing temporal behavior of the application code during execution* (e.g., col.8: 12-46)*; and*

identifying and replacing at least one repeated computational sequence in a bytecode stream with a semantically enriched opcode (e.g., col.9: 23 – col.10: 34).

Claim 23:

Sokolov discloses *the method of claim 1, further comprising: discovering at least one repetitive computational sequence used in a symbolic state* (e.g., FIG. 4, block 404, col.8: 1-11, while not end of stream (while still in loop), counting the number of times and checking whether it reaches/exceeds a predetermined number of times (a symbolic state)); *and*

generating a semantically enriched opcode corresponding to the at least one repetitive computational sequence used in the symbolic state (e.g., FIG. 4, block 408, col.11: 9-32, "Generate a Java macro instruction that represents the sequence of Java Bytecode instructions").

Claim 24:

Sokolov discloses *the method of claim 23, wherein the symbolic state comprises at least one of: control flow, data flow, entry points, and operational arguments* (e.g., FIG. 4, block 406 reaching a predetermined number of times? YES/NO (control flow selected between 2 branches), col.5: 55 – col.6: 26).

Claim 25:

Sokolov discloses *the method of claim 1, further comprising optimizing native code output by the virtual machine using a global optimizing compiler* (e.g., FIG. 4, col.7: 37-67, each application has different frequently repeated Java bytecode stream → block 408, different Java macro instructions are generated; FIG. 1A, col.8: 62 – col.9: 17, a compiler can be used with different Java macro instructions in different applications, i.e., a global optimizer compiler).

Claim 26:

Sokolov discloses *the method of claim 1, further comprising optimizing the virtual machine by offline compilation of the virtual machine by a host device* (e.g., col.3: 14 – col.4: 15; col.6: 66 – col.7: 19).

Claim 27:

Sokolov discloses *the method of claim 1, further comprising optimizing the virtual machine by online modification of a generic virtual machine by inserting a stub* (e.g., col.2: 64 – col.3: 9; col.7: 37-67),

the stub automatically loading a semantically enriched opcode when the virtual machine encounters an identified code segment with a bytestream (e.g., col.6: 66 – col.7: 19).

Claim 28:

Sokolov discloses *the method of claim 1, further comprising offline embedding of the semantically enriched opcodes in an application by substituting a semantically enriched opcode for a corresponding code segment* (e.g., col.9: 50 - col.10: 11).

Claim 29:

Sokolov discloses *the method of claim 1, further comprising online embedding of semantically enriched opcodes by a class loader* (e.g., col.2: 19 – col.3: 47),

said class loader substituting a semantically enriched opcode for a corresponding code segment (e.g., col.3: 14 – col.4: 15).

Claim 30:

Sokolov discloses *a system for optimizing performance of an interpreter based runtime system, the runtime system including a virtual machine, the system comprising:*

application code; an embedded processor; a virtual machine configured to translate said application code into native machine code compatible with said embedded processor (e.g., col.2: 19 – col.3: 47; col.3: 14 – col.4: 15)*;*

a detection module, said detection module being configured to analyze said application code to identify code segments that could be efficiently represented as semantically enriched opcodes (e.g., FIG. 2B, col.5: 55 – col.6: 26; FIG. 4, col.6: 66 - col.7: 19)*;*

at least a portion of the semantically enriched opcodes being specific to said application (e.g., FIG. 4, col.7: 6-19, block 404-406, each application has different frequently repeated bytecode → block 408, specific macro instructions are generated, i.e., specific semantically enriched opcodes)*;*

an embedding module, said embedding module being configured to embed said semantically enriched opcodes in said application (e.g., col.5: 66 – col.6: 9; col.8: 12-46; col.9: 23 – col.10: 34)*;*

a code generation module, said code generation module being configured to generate optimized action code for translating said semantically enriched opcodes according to symbolic states (e.g., FIG. 4, col.6: 66 – col.7: 19; FIG. 6A, "New" and "Dup"; FIG. 9A, block 902 and "Loop 1"; FIG. 9B, blocks 910 and 920; col.11: 9-32, Appendix A),

each of said symbolic states being represented by at least one symbolic variable (e.g., FIG. 4, block 406 "Sequence has been counted for at least a predetermined number of times? YES → block 408 "Generate a Java macro instruction that represents the sequence of Java Bytecode instructions", i.e., only generates macro instruction and passes it to Java interpreter based on whether a counter reaches/exceeds a predefined threshold (a predefined system state), wherein said predefined threshold (said predefined system state) is a variable and compared with a counter – see block 404 "Count the number of times …", emphasis added)*; and*

a build module configured create an application domain-specific virtual machine by incorporating said optimized action code and a bytecode set comprising said semantically enriched opcodes into said virtual machine (e.g., FIG. 5, col.7: 37-67; FIG. 7B-C, col.8: 62 - col.9: 17).

Sokolov does not explicitly disclose *performing a quantitative trade-off between execution time and memory space to determine effective semantically enriched opcodes and encoding the semantically enriched opcodes into interpreter action codes based upon the trade-off.*

However, in an analogous art, Egashira further discloses *performing a quantitative trade-off between execution time and memory space to determine effective semantically enriched opcodes and encoding the semantically enriched opcodes into interpreter action codes based upon the trade-off* (e.g., FIG. 3, col.7:60 – col.8: 53; FIG. 7, col.12: 57 – col.13: 18; col.6: 60 – col.7: 33).

It would have been obvious to a person having ordinary skill in the art at the time the invention was made to combine Egashira's teaching into Sokolov' teaching. One would have been motivated to do so to generate an object module file with a shorter execution time in a range of code size as suggested by Egashira (e.g., col.2: 45-54; col.3: 22-29; col.3: 62 – col.4: 39).

Claim 31:

Sokolov discloses *the system of claim 30, wherein said detection module is configured to analyze said application code using static optimization, said static optimization comprising parsing said application code to identify at least one repeated sequence of bytecodes* (e.g., col.5: 28 – col.6: 38; col.9: 23 – col.10: 34) *and*

replace said at least one repeated sequence of bytecodes with a semantically enriched opcode (e.g., col.5: 66 – col.6: 9; col.7: 37-67).

Claim 32:

Sokolov discloses *the system of claim 31, wherein said detection module is further configured to analyze said application code using dynamic optimization, said dynamic optimization comprising: analyzing temporal behavior of the application code during execution* (e.g., col.2: 64 – col.3: 9)*; and*

identifying and replacing at least one repeated computational sequence in a bytecode stream with a semantically enriched opcode (e.g., col.3: 14 – col.4: 15; col.5: 66 – col.6: 9).

Claim 33:

Sokolov discloses *the system of claim 30, wherein said generation module is further configured to: discover at least one repetitive computational sequence used in a said symbolic state* (e.g., col.8: 12-46)*; and*

generate a semantically enriched opcode corresponding to the at least one repetitive computational sequence used within said symbolic state (e.g., col.9: 23 – col.10: 34).

Claim 34:

Sokolov discloses *the system of claim 33, wherein said symbolic state comprises at least one of: control flow, data flow, entry points, and operational arguments* (e.g., col.5: 66 – col.6: 9; col.8: 1-11).

Claim 35:

Sokolov discloses *the system of claim 30, further comprising a global optimizer compiler configured to optimizing native code output by said virtual machine* (e.g., col.5: 28 - col.6: 38).

Claim 36:

Sokolov discloses *the system of claim 30, wherein said virtual machine is compiled offline by a host device* (e.g., col.8: 12-46).

Claim 37:

Sokolov discloses *the system of claim 30, wherein said virtual machine is modified online by inserting a stub, said stub automatically loading a semantically enriched opcode when said virtual machine encounters an identified code segment with a bytestream* (e.g., col.7: 37-67).

Claim 38:

Sokolov discloses *the system of claim 30, wherein said application code is modified offline by substituting a semantically enriched opcode for a corresponding*

code segment contained with said application code (e.g., col.3: 14 – col.4: 15; col.6: 66 – col.7: 19).

Claim 39:

Sokolov discloses *the system of claim 30, wherein a class loader embeds said semantically enriched opcodes online, said class loader substituting a semantically enriched opcodes for a corresponding code segment* (e.g., col.2: 19 – col.3: 47; col.5: 28 – col.6: 38).

(10) Response to Argument

1. Whether Claims 1 and 21-39 are unpatentable under 35 U.S.C. § 103(a) over Sokolov in view of Egashira (Brief, pages 9-14).

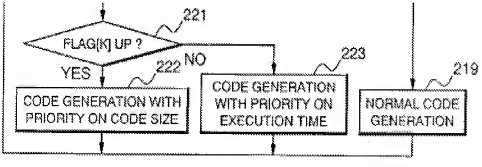
Claim 1 (Brief, pages 9-11)

Egashira fails to remedy the shortcoming of Sokolov in "*performing a quantitative trade-off between execution time and memory space* (code size) *to determine effective semantically enriched opcodes and encoding the semantically enriched opcodes into interpreter action codes based upon the trade-off*" (Brief, page 10).

Examiner respectfully disagrees with Appellants' arguments.

Egashira teaches *performing a quantitative trade-off between execution time and memory space* (code size) *to determine effective semantically enriched opcodes*:

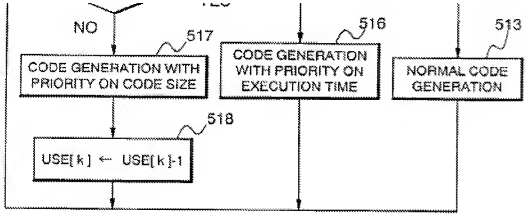
FIG. 3, col.7:60 – col.8: 53, block 221, using flag to indicate priority/trade-off between code size 222 ("memory space" as claimed) and execution time 223 → block 221/Yes, if flag is up, then code size ("memory space") has higher priority (222); block 221/No, if flag is not up, then execution time has higher priority (223) as illustrated (partial) in FIG.3 below:



partial FIG. 3

"and encoding the semantically enriched opcodes into interpreter action codes based upon the trade-off" as follows,

FIG. 7, col.13: 1-18, block 517, if code size ("memory space" as claimed) has priority, then code generation is carried out with priority on code size; block 516, if execution time has priority, then code generation is carried out with priority on execution time as illustrated (partial) in FIG.7 below:



partial FIG. 7

Claims 1 and 30 (Brief, pages 11-14)

Appellants further argued that Sokolov fails to teach or suggest "*statically embedding the semantically enriched opcode to optimize execution of the interpreter-based runtime system*" (Brief, page 11, emphasis added).

As an initial matter, examiner notes that the originally filed specification defined,

> "The sEc detection phase can be further subdivided into static or dynamic sEc detection stages (not shown in FIG. 2), both of which have the aim of deducing effective sEc-opcodes from a

> stream of Java bytecode which will produce an overall speedup in the execution of the Embedded Java runtime environment.
>
> In the static sEc detection phase, the particular compiled Java application is parsed for the most repetitive longest sequence of Java bytecodes. sEc bytecodes are selected from these based on cost and control flow criteria..." (specification, page 9, lines 10-16, emphasis added).

That is to say, what so-called and argued for "*statically*" process, here, is where the "compiled Java application" is parsed (bytecode stream 504 in Sokolov's FIG.5 is read/parsed) for detecting/selecting and *embedding* "the semantically enriched opcodes" (selecting and embedding 508 into 510, also in Sokolov's FIG.5) to optimize execution of the interpreter-based runtime system, i.e., that is to say, before runtime execution in a Java Virtual Machine ("statically embedding" before runtime execution – emphasis added).

In FIG. 5 (annotation added) below, Sokolov teaches Java Bytecode Verifier 500 performs a (single) "verification phase" (col. 6: 46-47 and 63-65):

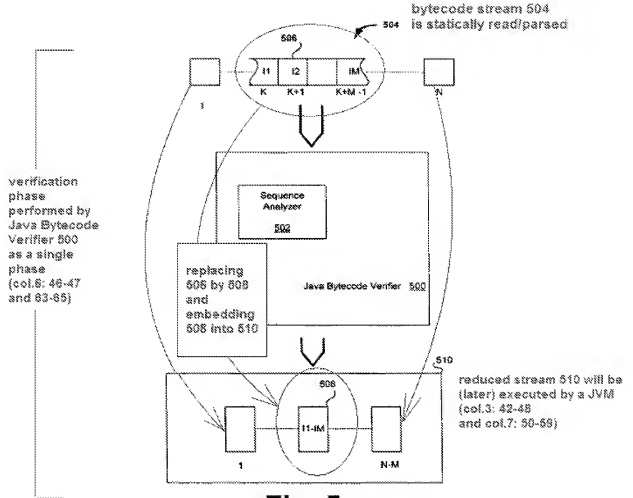


Fig. 5 (annotated)

col.7: 3-6 and 20-25, Java Bytecode Verifier 500 (statically) reads/parses Java bytecode stream 504 and detects bytecode instructions 506 (before runtime execution - statically detecting as set forth in the originally filed specification, page 9, lines 10-16 as extracted above);

col.7: 40-46, Java Bytecode Verifier 500 replaces bytecode instructions 506 including instruction block K - (K+M-1) by a macro instruction 508 (generating/encoding "semantically enriched opcodes" as claimed); and

Java Bytecode Verifier embeds said macro instruction 508 in output/reduced stream 510 ("statically embedding the semantically enriched opcode" as claimed).

Sokolov teaches Java Bytecode Verifier 500 reads/parses the input stream 504, embeds the macro instruction 508 ("semantically enriched opcodes") in a (single) verification phase (col.6: 46-47 and 63-65), and all said steps in the (single) verification phase are performed before runtime execution by a JVM (col.3: 42-48 and col.7: 50-59) – emphasis added. Thus, the step of embedding the macro instruction 508 into the reduced and/or optimized stream 510 is indeed a so-called "*statically embedding*" process and not a dynamic process as Appellants characterized of Sokolov teachings.

(11) Related Proceeding(s) Appendix

No decision rendered by a court or the Board is identified by the examiner in the Related Appeals and Interferences section of this examiner's answer.

For the above reasons, it is believed that the rejection should be sustained.

Respectfully submitted,

/Thuy Dao/

Examiner, Art Unit 2192

Conferees:

/Tuan Q. Dam/
Tuan Q. Dam
Supervisory Patent Examiner, Art Unit 2192

/Lewis A. Bullock, Jr./

Lewis A. Bullock, Jr.

Supervisory Patent Examiner, Art Unit 2193